Normalization of Java source codes

Léopold Ouairy, Hélène Le Bouder, Jean-Louis Lanet

Normalization of *Java* source codes

Léopold Ouairy[1], Hélène Le-Bouder[2], and Jean-Louis Lanet[3]

[1] INRIA, Rennes, France leopold.ouairy@inria.fr
[2] IMT-Atlantique, Rennes, France helene.le-bouder@imt-atlantique.fr
[3] INRIA, Rennes, France jean-louis.lanet@inria.fr

Abstract. Security issues can be leveraged when input parameters are not checked. These missing checks can lead an application to an unexpected state where an attacker can get access to assets. The tool *Chucky-ng* aims at detecting such missing checks in source code. Such source codes are the only input required for *ChuckyJava*. Since it is sensible to the identifier names used in these source codes, we want to normalize them in order to improve its efficiency. To achieve this, we propose an algorithm which works in four steps. It renames constant, parameter, variable and method names. We evaluate the impact of this renaming on two different experiments. Since our results are concluding, we show the benefits of using our tool. Moreover, we suggest another new way to improve *Chucky-ng*

Keywords: applet security, identifier renaming, Chucky-ng, Java Card

1 Introduction

An applet is a program embedded on *Java Cards*. Such applets contain secrets. Attackers can steal such secrets by exploiting wrong implementations of specifications. To do so, they try a fuzzing attack on the applet. This attack consists in sending many different messages to the card via the *Application Protocol Data Unit* (*APDU*) communication buffer. This *APDU* enables terminals to communicate with cards and vice versa. With this attack, attackers can detect and then exploit the wrong state machine of an applet. This exploit can lead the attackers to illegally obtain access to secret resources. This is possible if an applet omit to implement a check in a critical state, whereas the specification precises that such check shall be implemented. We want to protect applets by detecting those *missing-checks* in their source codes. Since applets are written in a subset of the *Java* language, we base our work on a version of *Chucky-ng* [3,8] adapted for *Java* source files. This version is entitled *ChuckyJava* and it directly works on source codes without the need to modify or annotate them. Such tool aims at detecting the *missing-checks* in applets by outputing an anomaly score between 1.00 and -1.00. If this anomaly score is 1.00, the applet forgets to perform a test. On the contrary, if the anomaly score is -1.00, then the applet is the only one to perform a test. Since there are false positives, an analyst is required in order to interpret *ChuckyJava*'s results.

We want to improve *ChuckyJava* by reducing the false positive rate and the entries of its report to analyze. Since it works with identifier names of constants, variables, parameters and methods, our idea is to normalize every identifier of an applet set. We suggest an algorithm to achieve this task and we need to develop the tool *IdentNormalize*. Such modifications can be adapted to other programming languages. However, our study focuses on *Java Card* applets. To benchmark our idea, we manually apply the modifications we recommend in the applet set. Then, we perform an analysis with *ChuckyJava* to verify if our idea improves the tool efficiency. Because our modifications are concluding, we suggest another idea to increase the *ChuckyJava* efficiency.

The context and state of the art are shown in section 2. Our contribution is presented in section 3. The results are explained in section 4. Our future work is shown in section 5. To finish, we conclude on our improvement in section 6.

2 Context and State of the art

2.1 How *Chucky-ng* operates

Chucky-ng's objective is to detect wrong implementations of source codes. To do so, it first parses source code within a project. Then the analyst chooses a function to process. *Chucky-ng* groups similar functions to the one selected. Finally, it compares this group by giving an anomaly score for every expression existing in the function set. This anomaly score's range is $[-1.00, 1.00]$. An anomaly score of -1.00 informs the analyst that the function executes an expression where none of the other do. On the opposite, a function with an anomaly score of 1.00 does not execute an expression whereas the others do. These expression are sensible to the name of identifiers. Two expressions not using the same identifier are flagged as not similar, even if the functions are semantically identical. *ChuckyJava* [4] is based on *Chucky-ng* but parses *Java* files instead of *C/C++*. Moreover, we use a layer above *ChuckyJava*: *FetchVuln*. This last one automates *ChuckyJava* by requesting an analysis for every methods of the file. It reports the vulnerable methods, according to *ChuckyJava* in a single output file.

2.2 Tool requirements

This section highlights the necessity for us to design a normalization tool. It has to normalize the variable, constant, parameter and methods names.

identifier names A specification of a *Java Card* applet precises the commands to implement and the expected behavior of an applet once it receives the command. Since *ChuckyJava* works partly with identifiers, it would work optimized if every applet implementing a specification uses the exact same identifiers for constants, fields and parameter names. However, our work shows two applets which implement the *OpenPGP 2.0.1* specification can use really different identifiers. For example, the command *PUT_DATA* is named *SET_DATA* in another applet. Even if both names have the same meaning, they are completely different for *ChuckyJava*. It would warn the user for an anomaly wich is a false positive. Our tool has to rename such constants as either *PUT_DATA* or *SET_DATA*.

method names Our tool has to normalize method names too. In *ChuckyJava*, there is a step where the tool gathers similar methods, using an unsupervised machine learning algorithm. The authors of *Chucky-ng*, the base of *ChuckyJava*, rely on a similarity of method names during its *neighborhood discovery* step. This step focuses on gathering the most similar methods (neighbors) within applet source codes, by using information retrieval techniques. Added to the *cosine distance* metric used, a distance reward bonus is added if the *Jaccard* distance between two method names are similar. The *Jaccard* distance measures the dissimilarity between two sets or two strings in our case. If the distance is closer to zero, then the strings are identical. On the opposite, if the distance is one, the strings have nothing in common. In our case, if they are not similar, a distance penalty is added to the method. As an example we have three methods *putData*. One of them forget to use a particular object of a meaningful type. *ChuckyJava* would not gather it as similar functions of *putData*, or even include a *getData* function. It would generate a lot of false positive. To prevent this, we want to force the gather of functions using the same names in order to compare methods implementing the same commands.

2.3 State of the art

There are different techniques or tools that aims at normalizing methods names in source codes. *INFOX* [9] is a tool that aims at tracking forks for a project on *github*. It then extracts features and clusters code snippets in order to highlight similar code.

Another approach is based on word study within methods [6]. The authors have based their work on different methods to analyze the similarity of their semantics. It can be achieved by using a *WordNet* object structure to organize the words. This structure organizes the words in a hierarchical way. Based on this, one can use a similarity measure such as the *Lowest Common Subsumer* (*LCS*) to determine if two words are related. Another technique uses the definition of words to determine if a pair of word is similar. This last one is called glossed based technique. However, the authors conclude by precising that none of the tools seem to perform well and require improvements.

The paper [2] uses *Latent Semantic Indexing* (*LSI*). This technique groups term frequencies of documents in matrices. The tool works by gathering similar vocabulary words from comments and identifier names. The technique is to firstly retrieve information within source codes. Then, it is able to cluster the code snippets and label them. The analyst gets a comprehensive view of the code snippet topics, without needing to search in it a particular information.

This plugin for the *Microsoft's Phoenix* framework [7] enables an analyst to detect clone detection. It is based on techniques to detect biological identical *ADN* sequences to search for perfect matches. It works by using suffix trees (*AST*) and uses identifier names. It creates *AST* of functions and compares them. There are other plagiarism detection techniques available on the internet to discover clones of code.

The requirements of our tool The tool which clusters topics by extracting words from comments and identifier could fit to our prototype. In a future work, we have to adapt it for working with methods as the base unit. This could be possible since applets in general are production ones. In other words, their source code should be clean and commented in most of the cases. The *Phoenix*'s plugin could match our requierements. It works only with perfect matches. However, the authors claim that for near exact matches, we should use the *Smith-Waterman* [1] algorithm. It uses matrices to determine the similarity of code snippets.

3 Contribution

We have considered the *Latent Semantic Analysis* (*LSA*) approach for the method normalization step. However, such method is based on similarity and its results may not be always be true. For the rest, we have made the choice to rename identifiers by deducing the ones which fit almost every applet of the set. The tool works in four steps we develop:

1. remove of unused variables,
2. constant and field names normalization,
3. method names normalization,
4. parameter names normalization.

1. Remove unused variables This step aims at removing unused variables. We work mainly on production applets. Some of them do not have any. However, it eliminates some useless output of *ChuckyJava*.

2. Constant and field names normalization *IdentNormalizer* tries to rename common constants within applet source codes. In most cases in specification implementation, the *short* constants defined at the beginning of the applets is the value of a specific command. This value may be unique. For example, the command *CHANGE_REFERENCE _DATA* has a value of *0x24* for its instruction byte. As it is declared once in an applet, we have a first step of comparison between constants of this value. We randomly choose the name for this constant by selecting a name in an applet and assigning it to the others. For constants using the same value at least twice in the source code should have different semantics, we prefer to not modify the names. We could replace all constants with their real values. However, *Chucky-ng* normalizes values as *$NUM*. For example, a *case 0x20* and a *case 0x30* would be transformed as *case $NUM*. By doing this, we would lose a lot of information. The tool normalizes the fields names too. For example, in the *OpenPGP* specification, there are three different *passwords* or *card holder verifications* which are *OwnerPin* objects. Some of them can be renamed as *ch1* or *pw1*. The trick here is to rename the identifiers by deducing their relations based on the object type *OwnerPin*. This technique renames identifiers regardless of any specification.

3. Method names normalization It is not trivial to find which methods are similar. One way to gather similar methods is to analyze their name with the *Jaccard* distance. However, if two functions are semantically identical but use different names, they would not

Algorithm 1: How *IdentNormalizer* operates

Data: source codes of applets S_1
Result: The applet modified S'_1
forall $s_1 \in S_1$ **do**
  remove of unused variables;
  constant and field names normalization;
end
V_1 := methods normalization names list;
forall $s_1 \in S'_1$ **do**
  method names normalization based on V_1;
  parameter names normalization;
end

be flagged as similar. If two names are similar but semantically different such as *setData* and *getData*, then they could be flagged as similar, even if they do not have the same objective. We want to compare functions with the same objective together. One way to solve our problem is to use information retrieval techniques (*IR*). It is possible to rely on latent semantic indexing or analysis (*LSI, LSA*) on source code and comments to extract the main topic of methods. Kuhn *et al.* [2] propose a technique entitled *Semantic Clustering*. It is based on both *LSI* and clustering. The information retrieval step extracts identifier names and comments from the source code. Then, a clustering method gathers code snippets of similar topics. With this technique, it is possible to group similar functions together in sets of topics in order to rename those methods. However, since this technique is based on statistics, it could contain classification errors.

4. Parameter names normalization At this point, the method names are normalized. This is an essential condition since we want to normalize parameter names. Even if function can use a different number of parameters, we rename the one that are used by every similar method names together. This is based on the type name. However, two functions can use twice the same type of object as parameter. In this case, within the function, it is possible to gather in a set the methods used by the first parameter and in another set the ones using the second parameter. By comparing those sets, it should be possible to determine if two are similar and should be renamed the same way. If the sets are identical, we suggest to leave the parameters names unchanged.

Summary of the operation Our tool *IdentNormalize* can be summed up as shown in Algorithm 1. It performs with a complexity of $O(2n)$ with n, the total number of files for all applets. However, the step of method names normalization may be more complex than $O(2n)$. The complexity of the tool depend on the *LSI* method's complexity.
Appendix A shows two different code snippets as S_1. Since they are semantically identical, our tool produces either two identical appendix A.1 or two appendix A.2 as S'_1. Only the class name remains unchanged.

4 Results

4.1 Description of the experiments

We have divided the experiments in two different sets. The first is experimented in a controlled environment. It contains one-file applets which perform a match-on-card algorithm. They have a similar structure and nearly the same number of methods. On the contrary, the second set is composed of four applets coming from different *github* sources. All of them implement the *OpenPGP v.2.0.1* specification [5]. However, their structure can be different and may interpret commands of the *APDU* in different ways. For each experiment, we present the result of *ChuckyJava* before and after the normalization method. Moreover, we compare only the results for the anomaly scores of -1.00 and 1.00. The reason behind this decision is because those numbers are the priority to focus on for an analyst.

4.2 Controlled environment experimentation

This set gathers eleven applets which were designed by students. Every applet is written within a single file. All of these applets are constructed with globally the same program structure. It focuses on the importance of the identifier normalization. From this set, we have divided the experiment in two experiments. We first perform a *ChuckyJava* analysis on the applets without any modification. Then, we apply our renaming algorithm on the set. We operate a second analysis on this new set. We present the impact of such a renaming tool on *ChuckyJava* by comparing before and after our renaming step as shown in Table 1. This same table shows only the results we have obtained for the anomaly scores -1.00 and 1.00. However, Fig 1. shows the values we get before and after renaming the identifiers for a few anomaly scores.

	number of entries for -1.00	number of entries for 1.00
original set	909	121
improved set	726	151

Table 1: Number of entries to analyse betwen the original set and the new one

We can see that the number of entries for the anomaly score -1.00 decreases on our custom set. However, it increases for the 1.00 anomaly score. After analysing at the results, it is because we have now the entries as anomalies for the *tests/cases* of *switches*. Such entries are lines which *ChuckyJava* reports to the analyst. Each line precises the anomaly score, the expression associated to this score and the location of the expression in its *Java* file. It adds a benefit for using a normalization tool.

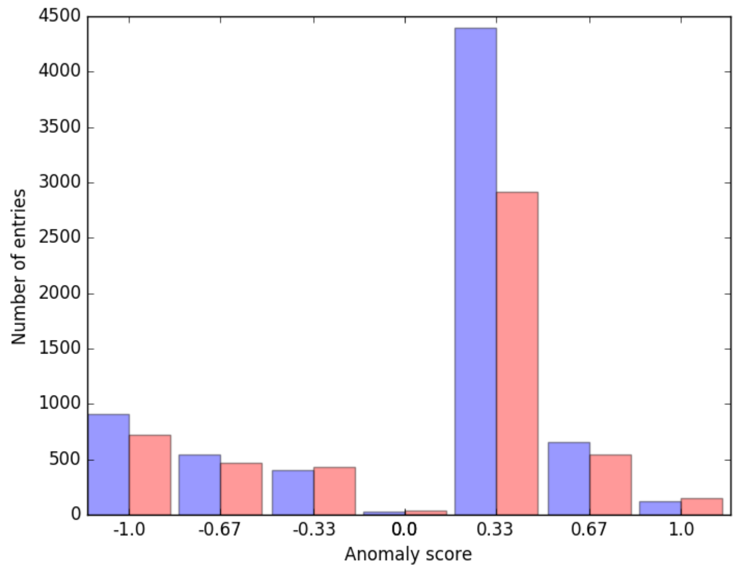

Fig. 1: The original set is on the left while the bar on the right represent the impact of *IdentNormalizer* in a controlled applet set

Conclusion *IdentNormalizer* has reduced the entries to analyze manually of roughly 15%. It corresponds to a decrease of 153 entries. This score can be obtained as the top performance because this set only focuses on the importance of the identifiers since the applet's structures have minor differences.

4.3 Second experimentation

We have gathered four applets implementing the *OpenPGP v2.0.1* specification. None of these applets come from the same author and have both different identifier names and program structures. Our applet set is composed of *MyPGPid*[4], *OpenPGPApplet*[5], *JCOpenPGP*[6], *Gpg*[7].
Fig 2. shows the result we have obtained from the first set to the last one which contains all the modifications. Table 2 summarizes the original number of results and with the improvements, for both -1.00 and 1.00 anomaly scores.

As for the controlled experiment, we can see that the number of entries for the anomaly score *decreases* for the anomaly score -1.00 but increase for the 1.00. To summarize

[4] https://github.com/CSSHL/MyPGPid
[5] https://github.com/Yubico/ykneo-openpgp
[6] https://sourceforge.net/projects/jcopenpgp/
[7] https://github.com/FluffyKaon/OpenPGP-Card

	number of entries for -1.00	number of entries for 1.00
original set	3769	255
improved set	3443	351

Table 2: Number of entries to analyse betwen the original set and the new one

the result, we lose roughly 6% of entries to analyse. It corresponds to a decrease of 230 lines. This result may seem to be a bit low. However, this is mainly because *ChuckyJava* is really sensible to the program structure. Indeed, it compares identifiers between applets to establish the anomaly score. However, since our applets are made from different authors and from different quality, we have concluded that the gain of 6% reflects the lower limit of this approach. For example, industrials may have structure pattern for the applets they implement, reducing the impact of such structure on the results of *ChuckyJava*.

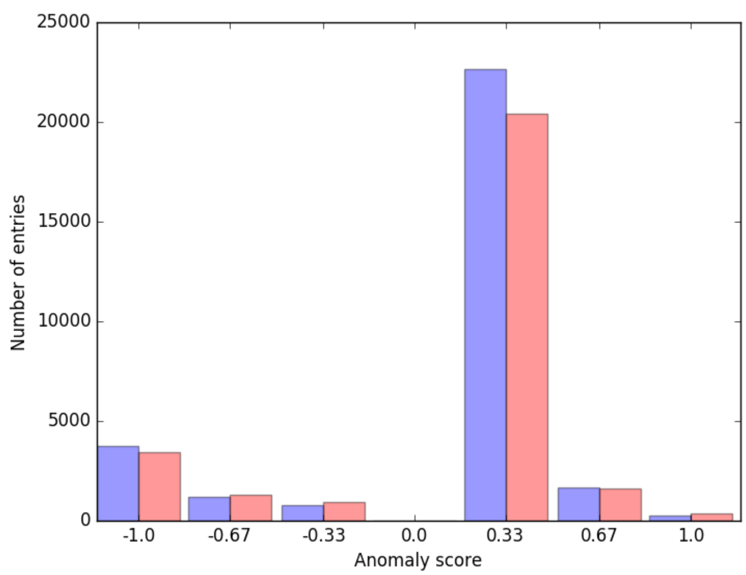

Fig. 2: The original set is on the left while the bar on the right represent the impact of *IdentNormalizer*, in a production applet set

5 Future work

This section presents the future work which completes the identifier normalization problem. We have noticed three different structural problems. The first one concerns an assignation problem. The second one focuses on the order of instruction within the source code. The last one is about the optional functionalities which an implementation offers.

5.1 Assignation type problem

We are aware of several assignation type problem. For example, the vast majority of the applets creates three different variables *pw1,pw2* and *pw3* for the three different *OwnerPin(s)*. However, one applet has been designed to store them in an array of *OwnerPin*. When a method of *OwnerPin* is called, the identifier name of the array is used instead of one of the *pw*. Because of this, *ChuckyJava* generates false positives. It reports this applet with an anomaly score of 1.00 since it is the only one to use such an identifier. This creates one additional output every time the program uses this array.

5.2 Instructions call order

We are aware of a structural problem. Listing 1.1 shows a code snippet representing an applet using the *APDU* buffer as a field. Every time the applet receives a command, it first retrieves the *buffer* before calling the corresponding method. On the opposite, 1.2 shows another applet retrieving the *APDU* buffer within each command method. However, even if those applets are semantically identical, they can be reported as anomaly. Because *ChuckyJava* analyzes methods as the base unit, it cannot see that the call to the *getBuffer* is made in another location in the source code. It adds additional false positive lines to the *ChuckyJava*'s report.

Listing 1.1: Snippet 1

```
byte[] buffer = new byte[APDU_MAX_SIZE];
protected void process(APDU apdu) throws ISOException{
        buffer = apdu.getBuffer();
        switch(buffer[ISO7816.OFFSET_INS]){
                case VERIFY:
                        verify(apdu);
                        break;
        [...]
}
private void verify(APDU apdu){
        //Some code, using without calling the APDU
}
```

Listing 1.2: Snippet 2

```
protected void process(APDU apdu) throws ISOException{
        byte[] buffer = apdu.getBuffer();
        switch(buffer[ISO7816.OFFSET_INS]){
                case VERIFY:
                        verify(apdu);
                        break;
        [...]
```

```
}
private void verify(APDU apdu){
        byte[] buffer = apdu.getBuffer();
        //Some code, using the buffer
}
```

5.3 Optional features

One last structure problem generating false positive is when a specification suggests optional features. As an example, the *OpenPGP v2.0.1* specification allows developers to use optional data objects (*DO*). If one applet decides to not implement them while the others do it, then the applet is flagged with many additional entries with an anomaly score of 1.00. On the opposite, if one option is implemented in only one applet, *ChuckyJava* generates entries with anomaly scores of -1.00. However, both the results are false positives since the implementation of the option is non mandatory.

6 Conclusion

We have developed *IdentNormalizer* which normalizes applet methods and identifiers names. It helps an analyst who uses *ChuckyJava* by reducing the number of entries between 6% to 15%. We have shown that the more the structure of an applet set is similar, the more *IdentNormalizer* is efficient. Our future work focuses mainly creating the tool that normalizes the identifiers. Moreover, we want to improve the efficiency of *ChuckyJava* by creating another tool which normalizes applet structures. As like *IdentNormalizer*, this new tool could be executed before performing an analysis with *ChuckyJava*.

References

1. Greenan, K.: Method-level code clone detection on transformed abstract syntax trees using sequence matching algorithms 4
2. Kuhn, A., Ducasse, S., Girba, T.: Semantic clustering: Identifying topics in source code 3, 5
3. Maier, A.: Assisted discovery of vulnerabilities in source code by analyzing program slices 1
4. Ouairy, L., Le-Bouder, H., Lanet, J.: Protection des systemes face aux attaques par fuzzing 2
5. Pietig, A.: Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems 6
6. Sridhara, G., Hill, E., Pollock, L., Vijay-Shanker, K.: Identifying word relations in software: A comparative study of semantic similarity tools 3
7. Tairas, R., Gray, J.: Phoenix-based clone detection using suffix trees 3
8. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: Exposing missing checks in source code for vulnerability discovery 1
9. Zhou, S., Stanciulescu, S., LeBenich, O., Xiong, Y., Wasowski, A., Kästner, C.: Identifying features in forks 3

A Renaming example

A.1 Code snippet 1

```
public static class Class01
{
        private final byte INS_VERIFY = (byte) 0x20;
        public void process(APDU apdu)
        {
                byte[] buffer = apdu.getBuffer();
                switch(buffer[ISO7816.OFFSET_INS])
                {
                        case INS_VERIFY:
                                verify(apdu);
                                break;
                        default:
                                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED)
                                    ↪ ;
                }
        }
}
```

A.2 Code snippet 2

```
public static class Class02
{
        private final byte INSTRUCTION_VERIFIES = (byte) 0x20;
        public void process(APDU a)
        {
                byte[] apduBuffer = a.getBuffer();
                switch(buffer[ISO7816.OFFSET_INS])
                {
                        case INSTRUCTION_VERIFIES:
                                verify(a);
                                break;
                        default:
                                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED)
                                    ↪ ;
                }
        }
}
```